

Domain-specific Languages in a Finite Domain Constraint Programming System

Markus Triska

Vienna University of Technology, Vienna, Austria,
 triska@dbai.tuwien.ac.at,
 WWW home page: <http://www.logic.at/prolog/>

Abstract. In this paper, we present domain-specific languages (DSLs) that we devised for their use in the implementation of a finite domain constraint programming system, available as `library(clpfd)` in SWI-Prolog and YAP-Prolog. These DSLs are used in propagator selection and constraint reification. In these areas, they lead to concise specifications that are easy to read and reason about. At compilation time, these specifications are translated to Prolog code, reducing interpretative run-time overheads. The devised languages can be used in the implementation of other finite domain constraint solvers as well and may contribute to their correctness, conciseness and efficiency.

Keywords: DSL, code generation, little languages

1 Introduction

Domain-specific languages (DSLs) are languages tailored to a specific application domain. DSLs are typically devised with the goal of increased expressiveness and ease of use compared to general-purpose programming languages in their domains of application ([1]). Examples of DSLs include *lex* and *yacc* ([2]) for lexical analysis and parsing, regular expressions for pattern matching, HTML for document mark-up, VHDL for electronic hardware descriptions and many other well-known instances.

DSLs are also known as “*little languages*” ([3]), where “little” primarily refers to the typically limited intended or main practical application scope of the language. For example, PostScript is a “little language” for page descriptions.

CLP(FD), constraint logic programming over finite domains, is a declarative formalism for describing combinatorial problems such as scheduling, planning and allocation tasks ([5]). It is one of the most widely used instances of the general CLP(·) scheme that extends logic programming to reason over specialized domains. Since CLP(FD) is applied in many industrial settings like systems verification, it is natural to ask: How can we implement constraint solvers that are more reliable and more concise (i.e., easier to read and verify) while retaining their efficiency? In the following chapters, we present little languages that we devised towards this purpose. They are already being used in a constraint solver over finite domains, available as `library(clpfd)` in SWI-Prolog and YAP-Prolog, and can be used in other systems as well.

2 Related work

In the context of CLP(FD), *indexicals* ([4]) are a well-known example of a DSL. The main idea of indexicals is to declaratively describe the domains of variables as functions of the domains of related variables. The indexical language consisting of the constraint “`in`” and expressions such as `min(X)..max(X)` also includes specialized constructs that make it applicable to describe a large variety of arithmetic and combinatorial constraints. GNU Prolog ([7]) and SICStus Prolog ([6]) are well-known Prolog systems that use indexicals in the implementation of their finite domain constraint solvers.

The usefulness of deriving large portions of code automatically from shorter descriptions also motivates the use of *variable views*, a DSL to automatically derive *perfect* propagator variants, in the implementation of Gecode ([8]).

Action rules ([9]) and *Constraint Handling Rules* ([10]) are Turing-complete languages that are very well-suited for implementing constraint propagators and even entire constraint systems (for example, B-Prolog’s finite domain solver).

These examples of DSLs are mainly used for the description and generation of constraint *propagation* code in practice. In the following chapters, we contribute to these uses of DSLs in the context of CLP(FD) systems by presenting DSLs that allow you to concisely express selection of propagators and constraint reification with desirable properties.

3 Matching propagators to constraint expressions

To motivate the DSL that we now present, consider the following quote from Neng-Fa Zhou, author of B-Prolog ([11]):

A closer look reveals the reason [for failing to solve the problems within the time limit]: Almost all of the failed instances contain non-linear (e.g., $X * Y = C$, $abs(X - Y) = C$, and $X \bmod Y = C$) and disjunctive constraints which were not efficiently implemented in the submitted version of the solver.

Consider the specific example of $abs(X - Y) = C$: It is clear that instead of decomposing the constraint into $X - Y = T$, $abs(T) = C$, a specialized combined propagator can be implemented and applied, avoiding auxiliary variables and intermediate propagation steps to improve efficiency. It is then left to detect that such a specialized propagator can actually be applied to a given constraint expression. This is the task of *matching* available propagators to given constraint expressions, or equivalently, mapping constraint expressions to propagators.

Manually selecting fitting propagators for given constraint expressions is quite error-prone, and one has to be careful not to accidentally unify variables that occur in the expression with subexpressions that one wants to check for. To simplify this task, we devised a DSL in the form of a simple committed-choice language. The language is a list of rules of the form $M \rightarrow As$, where M is a matcher and As is a list of actions that are performed when M matches a posted constraint.

More formally, a *matcher* M consists of the term $m_{\mathcal{C}}(P, C)$. P denotes a *pattern* involving a constraint *relation* like $\# =$, $\# >$ etc. and its arguments, and C is a *condition* (a Prolog goal) that must hold for a rule to apply. The basic building-blocks of a pattern are explained in Table 1. These building-blocks can be nested inside all symbolic expressions like addition, multiplication etc. A rule is applicable if a given constraint is matched by P (meaning it unifies with P taking the conditions induced by P into account), and additionally C is true. A matcher $m_{\mathcal{C}}(P, \text{true})$, can be more compactly written as $m(P)$.

<code>any(X)</code>	Matches any subexpression, unifying X with that expression.
<code>var(X)</code>	Matches a variable or integer, unifying X with it.
<code>integer(X)</code>	Matches an integer, unifying X with it.

Table 1. Basic building-blocks of a pattern

In a rule $M \rightarrow As$, each action A_i in the list of actions $As = [A_1, \dots, A_n]$ is one of the actions described in Table 2. When a rule is applicable, its actions are performed in the order they occur in the list, and no further rules are tried.

Figure 1 shows some of the matching rules that we use in our constraint system. It is only an excerpt; for example, in the actual system, nested additions are also detected and handled by a dedicated propagator. Such a declarative description has several advantages: First, it allows

$g(G)$	Call the Prolog goal G .
$d(X, Y)$	Decompose arithmetic subexpression X , unifying Y with its result. Equivalent to $g(\text{parse_clpfd}(X, Y))$, an internal predicate that is also generated from a similar DSL.
$p(P)$	Post a constraint propagator P . This is a shorthand notation for a specific sequence of goals that add a constraint to the constraint store and trigger it.
$r(X, Y)$	Rematch the rule's constraint relation, using arguments X and Y . Equivalent to $g(\text{call}(F, X, Y))$, where F is the functor of the rule's pattern.

Table 2. Valid actions in a list As of a rule $M \rightarrow As$

automated subsumption checks to detect whether specialized propagators are accidentally overshadowed by other rules. This is also a mistake that we found easy to make and hard to detect when manually selecting propagators. Second, when DSLs similar to the one we propose here are also used in other constraint systems, it is easier to compare supported specialized propagators, and to support common ones more uniformly across systems. Third, improvements to the expansion phase of the DSL benefits potentially many propagators at once.

```

1  m(integer(I) #>= abs(any(X))) => [d(X, RX), g((I>=0, I1 is -I, RX in I1..I))]
2  m(any(X) #>= any(Y))         => [d(X, RX), d(Y, RY), g(geq(RX, RY))]
3
4  m(var(X) #= var(Y)+var(Z))   => [p(pplus(Y,Z,X))]
5  m(var(X) #= var(Y)-var(Z))   => [p(pplus(X,Z,Y))]
6  m(any(X) #= any(Y))         => [d(X, RX), d(Y, RY)]
7
8  m(var(X) #\= integer(Y))     => [g(neq_num(X, Y))]
9  m(any(X) #\= any(Y) + any(Z)) => [d(X, X1), d(Y, Y1), d(Z, Z1),
10                                     p(x_neq_y_plus_z(X1, Y1, Z1))]
11 m(any(X) #\= any(Y) - any(Z)) => [d(X, X1), d(Y, Y1), d(Z, Z1),
12                                     p(x_neq_y_plus_z(Y1, X1, Z1))]
13 m(any(X) #\= any(Y))         => [d(X, RX), d(Y, RY), g(neq(RX, RY))]

```

Fig. 1. Rules for matching propagators in our constraint system. (Excerpt)

We found that the languages features we introduced above for matchers and actions enable matching a large variety of intended specialized propagators in practice, and believe that other constraint systems may benefit from this or similar syntax as well.

4 Constraint reification

We now present a DSL that simplifies the implementation of constraint *reification*, which means reflecting the truth values of constraint relations into Boolean 0/1-variables.

When implementing constraint reification, it is tempting to proceed as follows: For concreteness, consider reified equality ($\#=/2$) of two CLP(FD) expressions A and B . We could introduce two temporary variables, T_A and T_B , and post the constraints $T_A \#= A$ and $T_B \#= B$, thus using the constraint solver itself to decompose the (possibly compound) expressions A and B , and reducing reified equality of two *expressions* to equality of two finite domain *variables* (or integers), which is easier to implement. Unfortunately, this strategy yields wrong results in general. Consider for example the constraint ($\#<==>/2$ denotes Boolean equivalence):

$$(X/0 \#= Y/0) \#<==> B$$

It is clear that the relation $X/0 \#= Y/0$ cannot hold, since a divisor can never be 0. A valid (declaratively equivalent) answer to the above constraint is thus (note that X and Y must be constrained to integers for the relation to hold):

$$B = 0, X \text{ in } \text{inf}..\text{sup}, Y \text{ in } \text{inf}..\text{sup}$$

However, if we decompose the equality $X/0 \neq Y/0$ into two auxiliary constraints $T_A \neq X/0$ and $T_B \neq Y/0$ and post them, then (with strong enough propagation of division) both auxiliary constraints fail, and thus the whole query (incorrectly) fails. While devising a DSL for reification, we found one commercial Prolog system and one freely available system that indeed incorrectly failed in this case. After we reported the issue, the problem was immediately fixed.

It is thus necessary to take *definedness* into account when reifying constraints. See also [12], where our constraint system (in contrast to others that were tested) correctly handles all reification test cases, which we attribute in part to the DSL presented in this chapter. Once any subexpression of a relation becomes undefined, the relation cannot hold and its associated truth value must be 0. Undefinedness can occur when $Y = 0$ in the expressions X/Y , $X \bmod Y$, and $X \bmod Y$. Parsing an arithmetic expression that occurs as an argument of a constraint that is being reified is thus at least a ternary relation, involving the expression itself, its arithmetic result, and its Boolean definedness.

There is a fourth desirable component in addition to those just mentioned: It is useful to keep track of *auxiliary variables* that are introduced when decomposing subexpressions of a constraint that is being reified. The reason for this is that the truth value of a reified constraint may turn out to be irrelevant, for instance the implication $0 \neq \Rightarrow C$ holds for both possible truth values of the constraint C , thus auxiliary variables that were introduced to hold the results of subexpressions while parsing C can be eliminated. However, we need to be careful: A constraint propagator may *alias* user-specified variables with auxiliary variables. For example, in `abs(X) \neq T, X \geq 0`, a constraint system may deduce $X = T$. Thus, if T was previously introduced as an auxiliary variable, and X was user-specified, X must still retain its status as a constrained variable.

These considerations motivate the following DSL for parsing arithmetic expressions in reified constraints, which we believe can be useful in other constraint systems as well: A parsing rule is of the form $H \rightarrow Bs$. A head H is either a term $g(G)$, meaning that the Prolog goal G is true, or a term $m(P)$, where P is a symbolic pattern and means that the expression E that is to be parsed can be decomposed as stated, recursively using the parsing rules themselves for subterms of E that are subsumed by variables in P . The body Bs of a parsing rule is a list of body elements, which are described in Table 3. The predicate `parse_reified/4`, shown in Figure 2, contains our full declarative specification for parsing arithmetic expressions in reified constraints, relating an arithmetic expression E to its result R , Boolean definedness D , and auxiliary variables according to the given parsing rules, which are applied in the order specified, committing to the first rule whose head matches. This specification is again translated to Prolog code at compile time and used in other predicates.

<code>g(G)</code>	Call the Prolog goal G .
<code>d(D)</code>	D is 1 if and only if all subexpressions of E are defined.
<code>p(P)</code>	Add the constraint propagator P to the constraint store.
<code>a(A)</code>	A is an auxiliary variable that was introduced while parsing the given compound expression E .
<code>a(X,A)</code>	A is an auxiliary variable, unless $A == X$.
<code>a(X,Y,A)</code>	A is an auxiliary variable, unless $A == X$ or $A == Y$.
<code>skeleton(Y,D,G)</code>	A “skeleton” propagator is posted. When Y cannot become 0 any more, it calls the Prolog goal G and binds $D = 1$. When Y is 0, it binds $D = 0$. When $D = 1$ (i.e., the constraint must hold), it posts $Y \neq 0$.

Table 3. Valid body elements for a parsing rule

```

1  parse_reified(E, R, D,
2    [g(cyclic_term(E)) => [g(domain_error(clpfd_expression, E))],
3    g(var(E))           => [g((constrain_to_integer(E), R=E, D=1))],
4    g(integer(E))       => [g((R=E, D=1))],
5    m(-X)               => [d(D), p(ptimes(-1,X,R)), a(R)],
6    m(abs(X))           => [g(R#>=0), d(D), p(pabs(X, R)), a(X,R)],
7    m(X+Y)              => [d(D), p(pplus(X,Y,R)), a(X,Y,R)],
8    m(X-Y)              => [d(D), p(pplus(R,Y,X)), a(X,Y,R)],
9    m(X*Y)              => [d(D), p(ptimes(X,Y,R)), a(X,Y,R)],
10   m(X^Y)               => [d(D), p(pexp(X,Y,R)), a(X,Y,R)],
11   m(min(X,Y))          => [d(D), p(pgeq(X, R)), p(pgeq(Y, R)),
12     p(pmin(X,Y,R)), a(X,Y,R)],
13   m(max(X,Y))          => [d(D), p(pgeq(R, X)), p(pgeq(R, Y)),
14     p(pmax(X,Y,R)), a(X,Y,R)],
15   m(X/Y)               => [skeleton(Y,D,X/Y #= R)],
16   m(X mod Y)            => [skeleton(Y,D,X mod Y #= R)],
17   m(X rem Y)            => [skeleton(Y,D,X rem Y #= R)],
18   g(true)              => [g(domain_error(clpfd_expression, E))]]).
```

Fig. 2. Parsing arithmetic expressions in reified constraints with our DSL

5 Conclusion and future work

We have presented DSLs that are used in the implementation of a finite domain constraint programming system. They enable us to capture the intended functionality with concise declarative specifications. We believe that identical or similar DSLs are also useful in the implementation of other constraint systems. In the future, we intend to generate even more currently hand-written code automatically from smaller declarative descriptions.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4), 316–344 (2005)
2. Johnson, S. C., Lesk, M. E.: Language development tools. *Bell System Technical Journal*, 56(6), 2155–2176 (1987)
3. Bentley, J.: Little languages. *Communications of the ACM*, 29(8), 711–721 (1986)
4. Codognet, P., Diaz, D.: Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3) (1996)
5. Jaffar, J., Lassez, J-L.: Constraint Logic Programming. *POPL*, 111–119 (1987)
6. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. *Proc. Prog. Lang.: Implementations, Logics, and Programs* (1997)
7. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming (JFLP)*, Vol. 2001, No. 6 (2001)
8. Schulte, Ch., Tack, G.: Perfect Derived Propagators. *CoRR* entry (2008)
9. Zhou, N-F.: Programming Finite-Domain Constraint Propagators in Action Rules. *Theory and Practice of Logic Programming*, Vol.6, No.5, pp. 483–508 (2006)
10. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. *Special Issue on Constraint Logic Programming, J. of Logic Programming*, Vol 37(1–3) (1998)
11. Zhou, N-F.: A Report on the BPrologCSP Solver (2007)
12. Frisch, Alan M., Stuckey, Peter J.: The Proper Treatment of Undefinedness in Constraint Languages, *CP 2009, LNCS 5732*, 367–382 (2009)